The Bell System Technical Journal
Vol. 60, No. 6, July–August 1981
Printed in U.S.A.

No. 4 ESS:

Evolution of the Software Structure

By P. D. CARESTIA and F. S. HUDSON

(Manuscript received August 26, 1980)

This paper examines two different examples of how No. 4 ESS software has evolved through restructuring to meet the needs of the changing No. 4 ESS environment. The two software areas that underwent varying degrees of incremental restructure are Call Processing and Fault Recovery. We characterize the pre-restructure architectures, discuss the motivation and rationale which led to restructure, and present and evaluate the post-restructure architecture for each software system.

I. INTRODUCTION

No. 4 ESS, the largest-capacity electronic switching system ever developed by the Bell System, was developed to meet specific objectives of capacity and reliability.[1] To meet these objectives, the No. 4 ESS was designed using new hardware technology and a comprehensive stored program. The primary objectives of the initial No. 4 ESS program design were:

(*i*) real-time efficiency,
(*ii*) simple human interface,
(*iii*) defensive design,
(*iv*) ease of modification.

Since its initial service date, the No. 4 ESS has released a new generic software package approximately once a year incorporating major new hardware and software capabilities. Each new generic was built upon the previous generic. As the number of features provided by the No. 4 ESS grew, it became increasingly more involved in certain areas of software to accommodate new features without impacting existing features. The amount of time spent in regression testing had the potential for becoming an ever growing part of the software development interval, thus, increasing new feature development cost.

New software development methodologies that used top-down de-

sign and structured programming techniques gained wider use in the No. 4 ESS software development process. These rigorous approaches to software design effectively pointed out where certain areas of No. 4 ESS software could be improved.

A new high-level programming language, EPLX, was introduced that supported structured programming techniques and provided increased program readability, modularity, and maintainability.

Given the continuing demand for new features, the design objectives for software development had to be enhanced to place greater emphasis on ease of modification and flexibility to reduce the cost and development time of new system features. This increased emphasis along with new software development methodologies and programming languages led to a selective restructuring of areas of No. 4 ESS software, which were to be affected the most by new feature development.

The No. 4 ESS software areas which became major candidates for restructuring were call processing and fault recovery. Sections II and III give the restructuring process for these two software areas.

II. CALL PROCESSING RESTRUCTURING

To better understand the motivations and rationale for restructuring, we review the call processing architecture prior to restructure.[2] It should be made clear that the entire call processing system was not restructured. Instead, an incremental restructuring occurred which focused primarily on the task programs responsible for call handling actions. We discuss the task programs in light of their original design and their deficiencies. We give the motivation for and approach to restructure, along with a discussion and evaluation of the new architecture.

2.1 Call Processing before restructure

When the No. 4 ESS cutover in 1976, it provided the capability to interface with both local and toll switching machines, to function as a tandem and/or toll switch, and to interface with all of the trunk signaling types listed below:

(*i*) Dial Pulse (DP)
  (*a*) Delay Dial Start Dial
  (*b*) Immediate Start
  (c) Wink Start
(*ii*) Multifrequency (MF)
  (*a*) Wink Start
  (*b*) Delay Dial Start Dial
(*iii*) Common Channel Interoffice Signaling (CCIS)

The No. 4 ESS also provided the Centralized Automatic Message

Accounting (CAMA) function for trunks using dial pulse or MF address signaling.

The call processing programs were initially structured in a three-level hierarchy as shown in Fig. 1. The task dispensers (Level 1), which were entered directly from Executive Control, provided the interface for external stimuli received from the signaling hardware (Signal Processors and CCIS terminals) and the interface for internal stimuli received from timing and queuing programs. Executive Control provided both high- and low-priority entries to the task dispensers, which used the entries to poll the buffers in the signaling hardware for high- and low-priority reports and to determine if time-out conditions existed. If reports or time-out conditions existed, they were dispensed sequentially to the appropriate task program for processing. The task dispensers remained in control until all relevant external or internal stimuli had been processed or until an overload threshold had been reached that limited the amount of activity processed by the system during any base cycle.

The task programs (Level 2) performed the specific actions that switched calls. Task programs were entered from the task dispensers in response to a particular stimulus. The task program investigated

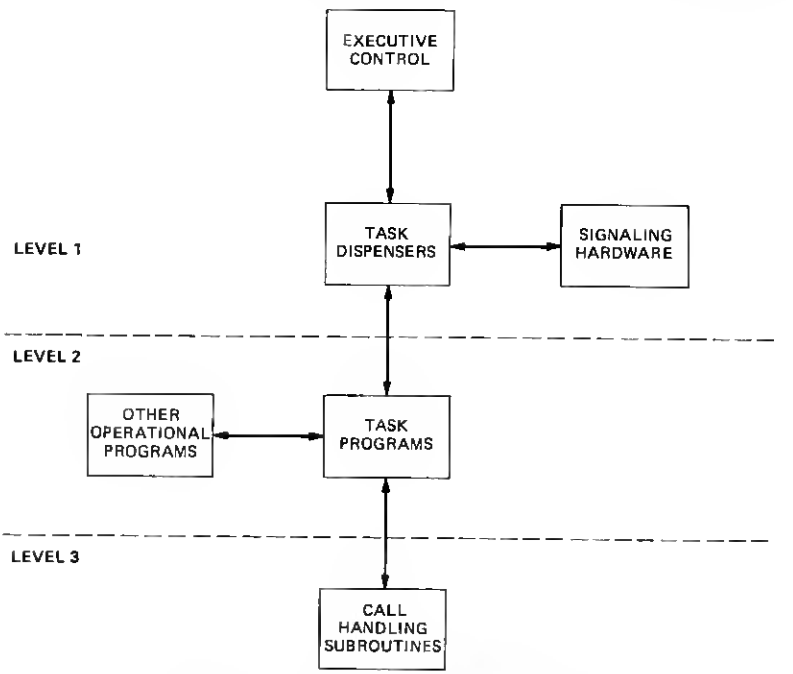

Fig. 1—Initial No. 4 ESS call processing architecture.

the present state of the call and, depending upon the present state and the stimulus, initiated the appropriate actions to advance the call to a new state. The present state of a call can be determined from the call register (CR) or trunk register (TR). The CR is a 64-word block of call store memory used for temporary storage of information during call setup. CRs are not dedicated on a per-trunk basis. Instead, there is an engineered number of CRs per office which are link-listed together. The TRs are two-word blocks of call store memory assigned on a per-trunk basis. TRs contain dynamic information about the current state of the trunk or call.

Certain repetitive or specialized call handling functions were designed as subroutines (Level 3) so they could be accessed by several task programs. Examples of call handling subroutines are seizing and initializing a CR, connecting incoming and outgoing trunks, hunting a service circuit, or pegging a traffic counter.

The task programs also interfaced with other operational programs during the processing of a call. These interfaces were established to allow independent software development of major operational functions such as audits, translations, network management, and trunk maintenance. Where these functions overlapped during the processing of a call, clearly defined interfaces were established with the task programs.

2.2 The Call Handling task programs

The Task Program block in Fig. 1 shows the set of task programs as shown in Fig. 2. The task programs were organized on a signaling-type/type-of-trunk basis and separated into incoming trunk and outgoing trunk programs. Each program was state driven and was responsible for acting on the stimuli dispensed from the task dispensers. The incoming trunk programs processed internal and external stimuli associated with the incoming trunk part of a call, and the outgoing trunk programs processed internal and external stimuli associated with the outgoing trunk part of a call. Internal stimuli were associated with events such as timing or queuing reports. External stimuli were physical trunk signals. Each incoming trunk program also had an interface with the Digit Reception and Analysis Programs, which were responsible for determining the outgoing routes for the call based upon the dialed digits. Digit sending to a large degree was part of the outgoing trunk programs.

The task program architecture arose primarily because of the means of communication with the signaling hardware. Communication with the signaling hardware was at a physical signal level (off-hook, on-hook) rather than a logical signal level (seizure, answer). Therefore, the signaling protocol for a specific trunk was required very early in

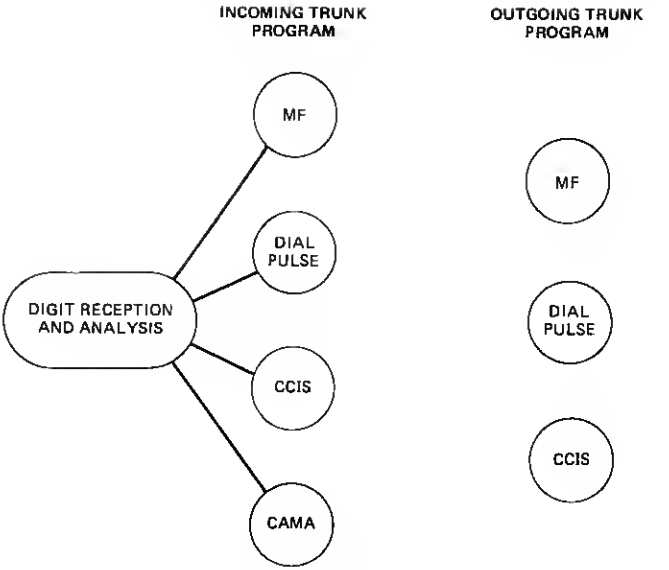

Fig. 2—Detailed view of task programs.

the processing of trunk signal reports. Rather than convert the physical signal into a logical signal prior to dispensing reports to task programs, the task programs were designed to handle the physical signals on a signaling-type/type-of-trunk basis. This approach resulted in a set of programs that were organized on an incoming and outgoing trunk basis for each signaling type/type of trunk.

Each program processed stimuli associated with its particular signaling type. However, there were always points in call setup where an incoming and outgoing trunk were involved in the call. They could be of the same or different signaling types. A stimuli at these stages of a call usually required actions by both the incoming and outgoing trunk programs. The design approach was to take one of two actions: (*i*) do whatever processing is required by the incoming trunk program, then pass control to the outgoing trunk program or vice versa, (*ii*) have the incoming or outgoing trunk program process the signal for both trunks associated with the call. The latter approach resulted in task programs that no longer contained processing logic for a single signaling type or for incoming or outgoing trunk. Incoming trunk programs made decisions based upon the type of outgoing trunk associated with the call and vice versa. For example, the CCIS task programs contained MF and DP signaling logic, etc. This approach was generally taken to save real time or to minimize program interfaces.

The drawbacks to such a task program design were: (*i*) proliferation

of decisions; (*ii*) duplication of program functions; (*iii*) dilution of program cohesion; and (*iv*) loss of independence between incoming and outgoing trunk. In some cases, task programs were call controllers and in others, single trunk controllers. The interfaces between incoming and outgoing trunk program became many and complex as shown in Fig. 3. The task program interfaces with the other operational programs further complicated the picture.

2.3 *Motivation for restructure*

With the development of the 4E3 generic for the No. 4 ESS, call processing was enhanced to provide the International Gateway Exchange Feature. This new feature required the addition of CCITT No. 5 and CCITT No. 6 signaling capabilities to call processing. Two new incoming and outgoing task programs were required along with the modification and retest of all existing task programs. Rather than add these signaling types to the existing architecture, thus further complicating an already complex structure, we considered restructuring the call processing task programs.

The goal of restructuring was to minimize the drawbacks of the current design, while at the same time, to minimize the effects of restructure upon the existing task programs which were known to be real-time efficient and virtually error-free. Eliminating duplication, strengthening program cohesion, and true separating incoming and

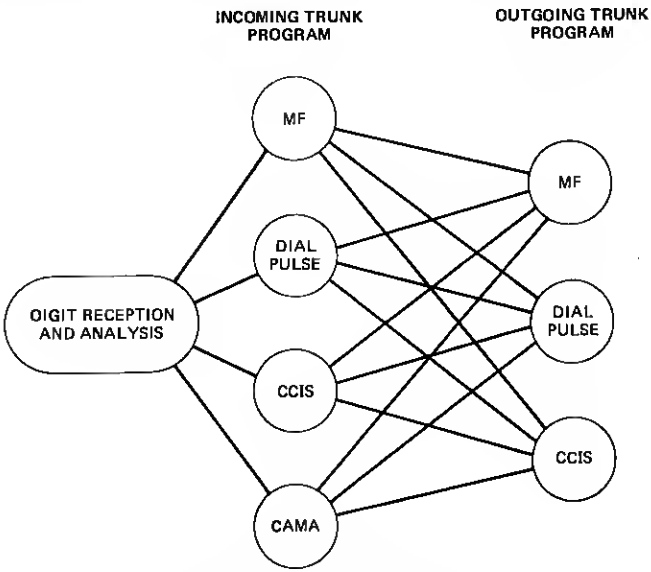

Fig. 3—Task program internal interfaces.

outgoing trunk processing were of special importance since future generics are very likely to require additional call handling task programs for new signaling types.

2.4 *Approach to restructure*

The key problem to be resolved with the restructure effort was how to make incoming trunk and outgoing trunk programs truly independent. The process began with formulating a universal call (shown in Fig. 4) and identifying the call events that must be processed to complete the call. At this point no effort was made to distinguish incoming trunk from outgoing trunk. The main focus was on the overall call. Seven call events were identified:

(*i*) Origination,
(*ii*) Digit reception,
(*iii*) Outgoing trunk selection,
(*iv*) Origination on the outgoing trunk,
(*v*) Digit sending,
(*vi*) Receive answer,
(*vii*) Receive disconnect.

The architecture began to materialize as a result of functionally decomposing the universal call into three sequential call stages:

(*i*) Setup—that part of the call from seizure on the incoming trunk through digit sending and connection of the incoming and outgoing trunks.

(*ii*) Post Setup—that part of the call during which time a voice path is connected, while awaiting answer and in the talking state.

(*iii*) Clearing—hardware and software trunk idling sequences after call termination.

The three sequential call stages were further decomposed into incoming trunk (ICT) and outgoing trunk (OGT) processes, resulting in the functional decomposition shown in Fig. 5.

The final phase of the process addressed the basic problem of isolating ICT and OGT processing. A new program function was created to consolidate the communication interfaces between ICT and OGT task programs and to oversee common call related functions. This program was called the Report Dispenser. Its inclusion in the new architecture made it possible to remove from the task programs any trunk signaling logic dealing with the other trunk involved in the call and to create trunk handlers.

The Report Dispenser was the single most important addition to the call processing architecture, because it introduced the use of logical signals as the means of communication between trunk handlers. Trunk handlers could now communicate with the Report Dispenser without involving another trunk handler. Incoming trunk and outgoing trunk

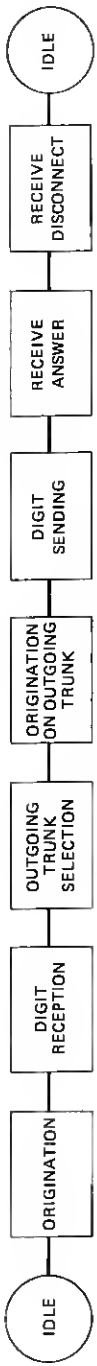

Fig. 4—Universal call flow diagram.

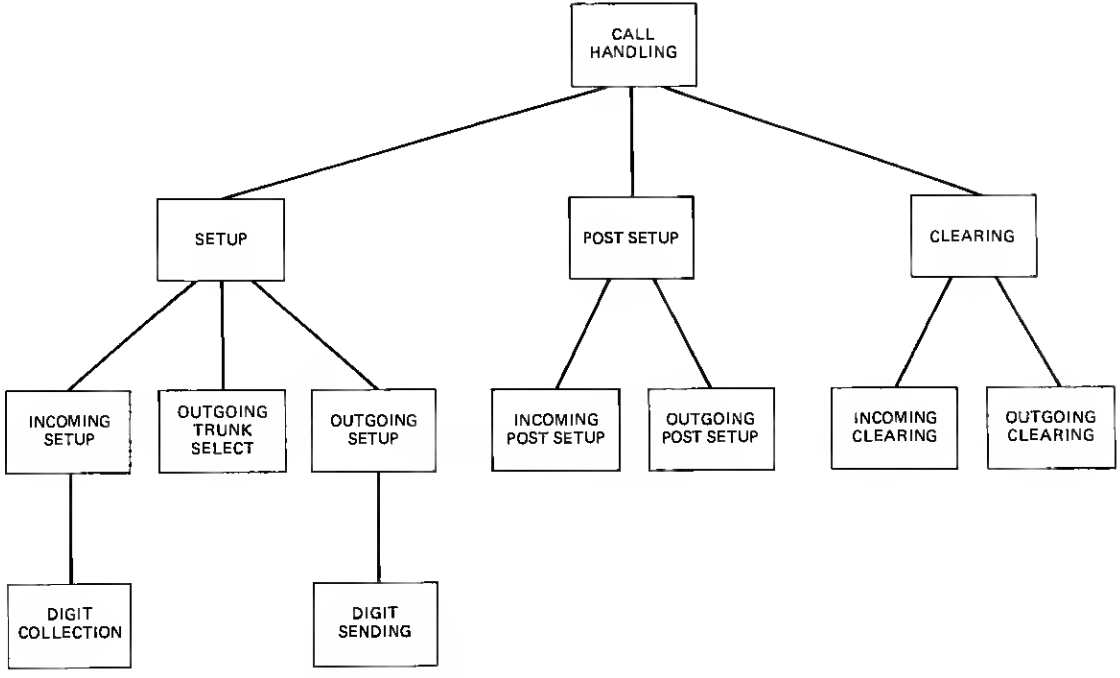

Fig. 5—Functional decomposition of a universal call.

processing became independent. The complex interface between incoming and outgoing trunk programs had been replaced with a standardized interface that used logical signals as a means of communicating with a central point of control wherein call-related decisions requiring knowledge of the other trunk were made. The new architecture became one where task programs/trunk handlers communicated with trunk circuitry via physical signals and communicated with other trunk handlers via logical signals.

2.5 New architecture overview

The primary features of the new architecture are as follows:

(*i*) The splitting of call handling into parallel real-time processes (finite state machines), which control states of the incoming trunk, the outgoing trunk selection process, and the outgoing trunk.

(*ii*) The consolidation of communication decisions, which link these finite state machines in a program called the Report Dispenser.

(*iii*) The identification of a subset of call handling functions that can be implemented as subprocesses (submachines) under control of incoming or outgoing trunk handlers. These functions were common to most calls and relatively independent of signaling type. They are digit reception and digit sending.

An architecture based upon trunk handlers is advantageous from the standpoint of minimal impact upon the existing call processing task programs. The basic logic of the task programs can be maintained; the changes are limited to separating incoming and outgoing trunk functions, eliminating redundant code, and interfacing the task programs with the Report Dispenser. Figure 6 illustrates the major modules in the new architecture and the control hierarchy.

All task dispenser reports are made on a trunk state basis. This means that report dispensing is based strictly on the state of one trunk involved in a call to the trunk handler responsible for handling that type of trunk.

All internal and external stimuli are dispensed by the task dispensers in the same manner as existed in the pre-restructure system. A new task dispenser was added as part of the restructure effort to interface with the CCITT No. 6 signaling terminal.

When a trunk handler receives a physical signal from the task dispenser it takes whatever action is appropriate and then reports a logical call event to the Report Dispenser. The Report Dispenser determines the next call action to initiate based upon the logical event and may invoke per call common functions, such as outgoing trunk selection, or invoke the other trunk handler involved with the call. When the signal has been completely processed, control is returned to the task dispenser via the Report Dispenser and the trunk handler

that initially received the stimulus. In summary, a physical signal is passed to the trunk handler, which converts the signal to a logical signal (answer, disconnect, etc.) based upon the state of the trunk. The logical signal now becomes the stimulus to the Report Dispenser to stimulate further call processing actions.

The block called Common Call Functions in Fig. 6 consolidates many common call related functions and interfaces which in the pre-restructure architecture were spread throughout the task programs. Many of the interfaces with the other operational programs are consolidated here.

The Digit Reception and Digit Sending functions appear as submachines under the incoming and outgoing trunk handlers. They are programs which are invoked by the trunk handlers and are logical rather than physical signal driven. Task dispenser reports are directed to these submachines and not to the trunk handlers. This allowed for efficient real-time execution in the processing of these reports and

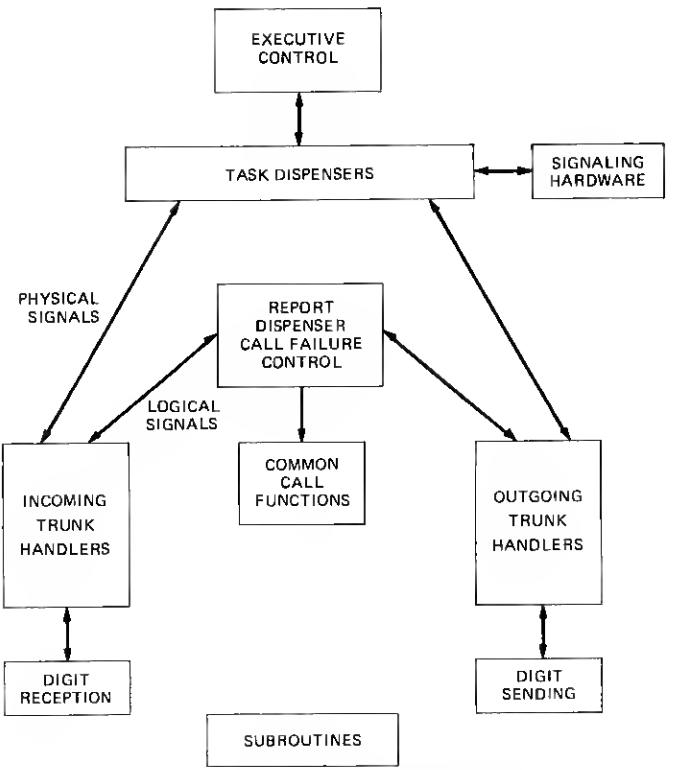

Fig. 6—No. 4 ESS call processing architecture.

does not burden the trunk handlers with detailed knowledge of how the submachine performs its function.

Call Failure Control has the same relative position in the new architecture as the Report Dispenser and is responsible for controlling the clearing of incoming and outgoing trunks as a result of Ineffective Attempts, i.e., calls that are not successfully completed.

2.5.1 A simple call

To more clearly understand the structure, interfaces, and control, we describe a simple DP-to-DP call. We incorporate only those events needed to successfully complete the call because the picture becomes more complicated when call anomalies are taken into consideration. The scenario is based upon the universal call diagram. Figure 7 represents the call flow diagram for the call. Incoming Trunk actions are represented along the top horizontal axis. Logical events are reported to the Report Dispenser, which then communicates these call events to the OGT. Outgoing trunk actions are represented along the bottom horizontal axis. The call actions progress in sequence from left to right.

The call begins with the receipt of an off-hook origination on the idle ICT. The physical off-hook signal is passed from the task dispenser to the ICT handler which prepares for digit collection. When the ICT is ready to receive digits, an integrity check signal is sent backward toward the originating office. The next ICT action is to receive digits. This action is performed by the Digit Reception Program. The Digit Reception Program also analyzes the digits to determine the outgoing trunk group for the call. The Report Dispenser is now notified that the call is ready for OGT selection. The Report Dispenser invokes the OGT selection program which is a common call function. After a successful return from the OGT selection program, the Report Dispenser invokes the OGT handler. The first OGT handler action is to seize the OGT. After seizing the trunk, the Report Dispenser is informed that seizure is complete. For this call, no ICT action is required at this point. Action is required if the ICT is CCIS or CCITT No. 6. This knowledge resides only within the Report Dispenser. The OGT handler waits for receipt of the integrity check signal from the far end office indicating readiness to receive digits. The OGT handler invokes the Digit Sending program, which deletes or prefixes digits to the dialed number and controls the outpulsing process on the OGT. When all digits have been outpulsed, control passes back to the Report Dispenser indicating the outgoing part of the call is complete. At this point, the call moves from the setup stage to the post setup stage. The CR is released and the voice path between ICT and OGT is completed. Both actions are common call functions. Each trunk handler places itself in the waiting-for-

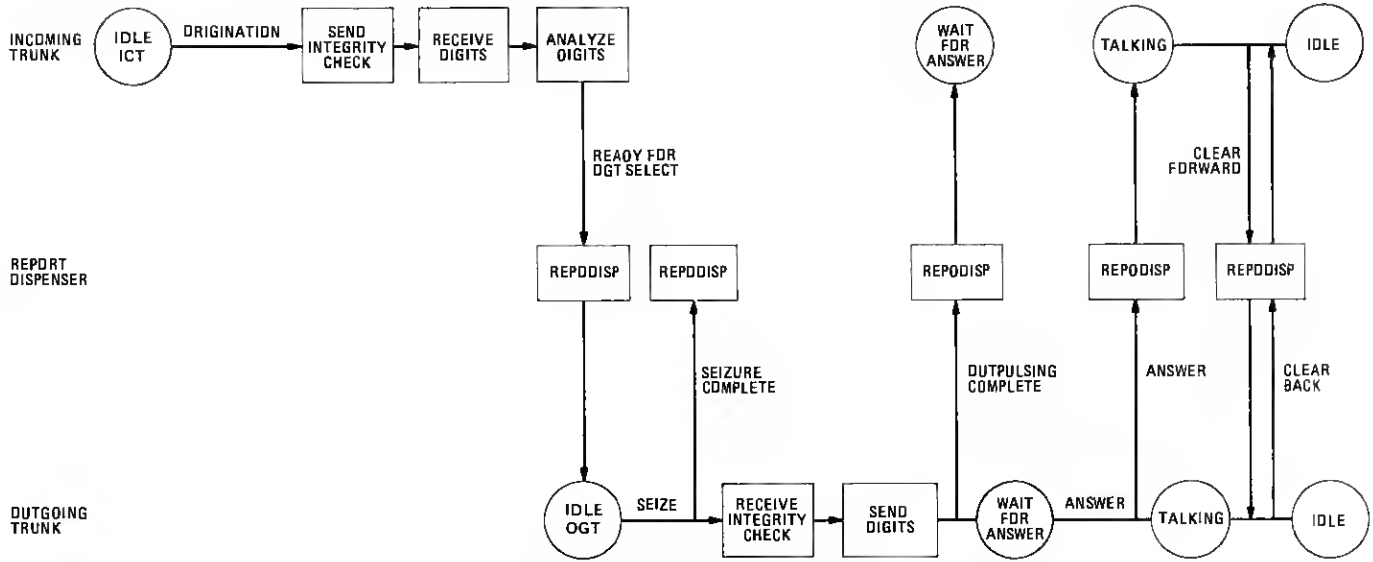

Fig. 7—Dial pulse-to-dial pulse call flow.

answer state. The next signal the No. 4 ESS expects to see is off-hook answer on the OGT or on-hook clearforward on the ICT, should the originator disconnect. In the case of the answer signal, the OGT handler processes the signal for the OGT and reports the logical event to the Report Dispenser, which in turn, passes the event to the ICT handler for processing. The next event in the call will be an on-hook disconnect on either trunk.

If the ICT receives an on-hook clearforward, the ICT handler reports the event to the Report Dispenser which invokes the ICT and OGT clearing routines to idle the trunks. If an on-hook clearback signal is received by the OGT, the OGT handler passes the event to the Report Dispenser, which invokes the ICT handler to send a clearback signal on the ICT. A clearback does not cause the call to be idled. A clearforward must be received to idle the call.

This simple example of the DP-to-DP call demonstrates how the Report Dispenser isolates the ICT and OGT from having knowledge of the other. We can then extend this example to cases where the ICT and OGT are of different signaling types and show that by communication with the Report Dispenser using logical call signals any type of ICT can interwork with any type of OGT, given the necessary logical signals have been defined.

2.5.2 The report dispenser

Communication between the trunk handlers is consolidated in the Report Dispenser. When a trunk handler detects a logical event that may be significant to the other trunk handler on that call, it reports that event to the Report Dispenser. The Report Dispenser determines the other trunk handler on the call and passes control to the appropriate trunk handler. This consolidation of what are primarily signaling type decisions about the other trunk handlers involved in the call results in an overall reduction of code and simplifies the addition of new signaling types. The addition of a new signaling type to this call processing system obviously involves the design and development of an ICT and OGT handler. However, if the new signaling type does not require the addition of any new logical signals, then the Report Dispenser only requires slight modification to include the new signaling type.

The trunk handlers communicate with the Report Dispenser by means of logical signals and pass additional data with the CR and TR. The signals are divided into two categories: setup and post setup. Setup signals are passed to the Report Dispenser, along with the CR, during the setup stage of a call. The signaling type of the ICT and OGT and the state of the call are stored in the CR. Based upon CR data and the logical signal, the Report Dispenser makes the decision on what to do next in the processing of the call. Post setup signals are passed to

the Report Dispenser in the post setup stage of the call, which is after the CR has been released. The Trunk Scanner Number (TSN), which identifies the ICT or OGT, is passed along with the logical signal. Through data translations using the TSN, the signaling type of the trunk and the TR are found. Based upon the signaling type of the trunk and the logical signal, the proper next step in the call can be taken. This mechanism then allows ICT and OGT handlers to have no knowledge about the other trunk involved in the call. That knowledge, along with the knowledge of the state of the call, resides within the Report Dispenser.

There are several functions in call processing that are call-event dependent and signaling-type independent. The Report Dispenser provides a central point for calling procedures associated with these common call functions. The common call functions include:

(*i*) Voice path setup and take down.

(*ii*) Interfacing with the No. 4 ESS Service Observing System, if it is active on a call.

(*iii*) Interfacing with the No. 4 ESS Network Management Programs.

(*iv*) Call register release.

(*v*) Interfacing with the No. 4 ESS Inward Wide Area Telecommunications Service Billing Program.

(*vi*) Interfacing with the No. 4 ESS Call Detail Recording Program for international calls.

The advantages of calling common call functions from the Report Dispenser instead of assigning that responsibility to the trunk handlers are as follows:

(*i*) minimization of errors—the functions can be called from a single module;

(*ii*) reduction in real time—such functions generally required the knowledge of both trunks, information which the Report Dispenser had available but would have to be regenerated in a trunk handler;

(*iii*) elimination of code—the functions can be called from a single module;

(*iv*) simplification of changes or additions to event-dependent functions—a significant advantage when adding features that are signaling-type independent.

2.5.3 Call failure handling

Call failure handling or final handling is the name given to the cleanup process for calls that fail to complete in a normal manner. Calls can fail due to machine error (hardware or software), customer error (misdialing, early abandon), or network conditions (congestion, network management controls).

There is a general class of events in the No. 4 ESS known as call

irregularities, which cause either a retrial attempt, or an abnormal termination of the call. An abnormal termination is called an ineffective attempt. Most ineffective attempts are because of an inability to complete the setup stage of a call. Some examples are as follows:

(*i*) ICT abandon in the setup stage.

(*ii*) Network congestion (all circuits busy, network management controls).

(*iii*) Failure on retrial attempts (glare, outpulsing errors, integrity check failures).

(*iv*) Office congestion (no CRs or service circuits, network blockage, overload controls in effect).

Some ineffective attempts occur in the post setup stage, such as loss of transmission (carrier failure).

Final handling clears ineffective attempts, allowing call processing resources (CR, trunks, service circuits) to be reused for new calls. Announcements and tones are also provided to help inform the customer of the situation.

There are numerous states that a call could be in when final handling is required. A call could be using many combinations of machine resources (i.e., CR timing lists, service circuits). Rather than determine the exact state of a call and idle only those resources and processes associated with that state, final handling checks for and idles all possible resources and processes on a call. In this way, calls can be cleared that have invalid states or invalid resources associated with valid states.

Final handling can be thought of as having two components, a call failure controller and a set of trunk clearing modules. The call failure controller holds a position in the architecture equivalent to the Report Dispenser, and like the Report Dispenser performs functions associated with common call related facilities (see Fig. 6). The trunk clearing modules are part of each trunk handler and provide a customer treatment based upon the trunk signaling type.

The call failure controller could have been made part of the Report Dispenser and final handling conditions treated via the same logical event-type interface that trunk handlers have with the Report Dispenser. However, the call failure controller already existed in the pre-restructure architecture and changing this interface would have had a major impact on the existing trunk handlers. A logical event-type interface like that of the Report Dispenser was provided in the call failure control module to accommodate the CCITT No. 5 and CCITT No. 6 trunk handlers, since they were new task programs to be developed during restructuring.

When a call requires final handling, the trunk handler interfaces with the final control module, which clears common facilities and

invokes the particular trunk clearing modules to idle remaining trunk-related facilities and to provide proper customer treatment for the call.

2.6 Evaluation

The interfaces and direction of communication between the trunk handlers, the task dispensers, and the Report Dispenser have become call processing programming standards. In some cases these standards produce a call flow which sacrifices real-time efficiency for the sake of uniformity. However, the sacrifice of real time is justified to maintain the integrity of the architecture. The analysis of call processing program errors and the changes required for program correction are a much simpler task because of easier problem isolation. The architecture makes the addition of new signaling types and design changes a more quantifiable job. The placement of new modules becomes readily apparent in the structure because the architecture directs the designer to a specific process of functional decomposition. The new signaling type is separated into ICT and OGT processes. Each process is then further decomposed into setup, post setup, and clearing functions, and new logical signals, if any, are identified. This process to a degree forces a consistent approach to the first level of task program modularity.

Since the call processing restructure was incremental, major portions of the existing code were not redesigned or rewritten. Existing task programs were not totally reorganized into distinct setup, post setup, and clearing modules. However, additional reorganization continues as new features are added to the call processing task programs. The implementation of the new CCITT No. 5 and CCITT No. 6 trunk handlers followed the call processing program standards completely. In addition, CCITT No. 6 was implemented with the use of EPLX.

As part of the 4E5 generic, the Mass Announcement System (MAS) feature was added to the No. 4 ESS. The MAS feature required a number of new types of calls to be processed by the No. 4 ESS and was a major software development undertaking in call processing. Using the structure of the new architecture as the basis for MAS feature decomposition and design, changes were made to add MAS to the call processing system. The feature addition was successful. Many of the new MAS calls executed correctly soon after introduction for testing in the system laboratory environment. At the same time, the old call types that the No. 4 ESS previously accommodated remained intact with no errors introduced as a result of the MAS feature addition.

The restructuring effort did not go without problems, the prime being increased real-time usage. After the architecture was solidified and much of the software developed, certain real-time critical parts were reviewed and optimized until real-time performance was judged to be within reason. As real-time improvements were made to the

EPLX, additional changes were made to certain areas of call processing to further improve real-time performance.

III. MAINTENANCE SOFTWARE RESTRUCTURING

Peripheral maintenance software for the No. 4 ESS also has been selectively restructured to minimize the cost of developing such software and provide the ability to continue to add new hardware features. The restructured peripheral fault recovery system incorporates operating system concepts, top-down hierarchically designed control structures, and use of a formal development methodology. This section gives a brief overview of the pre-restructured maintenance system. We also give error recovery and system recovery concepts, the motivation behind restructuring a selective part of the maintenance system, and finally a description of the restructured system, and an evaluation of the benefits of the new system.

3.1 Maintenance system overview

The stringent reliability and maintainability requirements of the No. 4 ESS affect both the hardware and software design of the system. In the software, we have developed a large program package to provide maintenance functions.[3] This maintenance software package consists of four functional areas that play an essential role in providing the maintenance capabilities of the No. 4 ESS: (*i*) fault recovery; (*ii*) diagnostics; (*iii*) system reinitialization and recovery; and (*iv*) system integrity and audits. Fault recovery is concerned with the system recovery from hardware faults. Diagnostics aid the craftperson in the identification of faults and repair of a faulty unit. System reinitialization is concerned with the overall coordination of system recovery from multiple or severe hardware and software malfunctions. The system integrity area is concerned with the detection of and recovery from memory mutilation. These software areas are designed based on specific error recovery and system recovery concepts. We give these concepts in later sections for background information.

Various types of redundancy (e.g., duplex, $n + 1$ duplication, $n + 2$ duplication, and load sharing) are used in the different subsystems to meet hardware reliability requirements. Each subsystem uses a number of error-detection techniques such as parity, matching, order acknowledgment, and self checking. These hardware characteristics place specific requirements on the maintenance software, particularly the fault recovery area, which is tightly coupled to hardware design.

3.1.1 Error recovery concepts

The maintenance software is built around several levels of execution based upon both maintenance software and hardware error detection triggers. Table I shows the maintenance program execution levels in

Table I—Maintenance program execution levels

Layer	Level	Function
System Recovery	Phase 4 Phase 3 Phase 2 Phase 1	System initialization
Maintenance Hardware Interrupts	A B C D E	Processor fault recovery
	F	— Peripheral recovery
	G	— Utility and timing
	K	— Segment timing validation
Maintanenace Software	Interject BLM	Fault recovery
Manual requests Audits Diagnostics	Base	Low-priority task

the No. 4 ESS. Only those execution levels which are applicable to peripheral fault recovery are discussed. The remaining levels of execution are presented in Ref. 4 on the 1A Processor.

Base level is the lowest and the normal level of system execution. All the call processing work described earlier, as well as audits and diagnostics, are normally executed at this level. Base level maintenance (BLM) is the next level and is triggered by defensive checks provided in software or firmware. Interject level is the next higher level and is guaranteed to be served by the 1A Processor every 10 ms. F-level interrupts report peripheral errors and are of two types: peripheral unit failure (PUF) and autonomous peripheral unit failure (APUF). The PUF interrupt is generated by the 1A Processor when a peripheral frame fails to acknowlege, or incorrectly acknowledges a directed order. The APUF interrupt is generated autonomously by a peripheral unit failure. The 1A Processor scans for APUFs every 11.2 μs. Base level maintenance, interjects, and both types of F-level interrupts invoke peripheral fault recovery. Fault recovery actions can also be stimulated by manual requests, such as input messages or power control switch requests.

3.1.2 System recovery concepts

When fault recovery succeeds in reconfiguring the system so the faulty unit is not in service, repair activity commences. However, in cases when complex or multiple faults prevent fault recovery from configuring an acceptable working system, system recovery actions are taken. Phase recovery is the highest level system recovery action and can be initiated either manually or by software. Phase recovery can

escalate through four phases, where phase 1 is the least severe and phase 4 is the most severe. Phase 4 can only be requested manually.

There are two types of Phase 1s. The first type executes a specified set of audits that correct data mutilation. The second type is a directed phase 1 and is initiated by a fault recovery action on a peripheral unit which results in a loss of service provided by the unit. The directed phase 1 initializes software structures associated with tbe faulty unit. A phase 2 initializes additional software structures and also performs a unit access test on the peripheral hardware when it is initiated by F-levels. Phase 3 is the highest level phase that is automatically requested. It performs additional software structure initialization and additional tests on the hardware. A phase 4 performs a total system initialization and can only be requested manually.

3.2 Motivations for a modern structure

3.2.1 Drawbacks of original implementation

Since the initial generic release (termed 4E0), each new generic has included new features, hardware cost reductions, and enhancements. Each generic must continue to meet the original design objectives of the system for capacity and reliability and, at the same time, provide new services, and take advantage of the rapidly changing technology through bardware cost reductions. By the end of the second generic (4E1), the continuing demand for new hardware features and cost-reduced hardware was evident. We, therefore, determined how the development cost could be reduced for maintenance software. Of the four maintenance software areas described earlier, we found the fault recovery area to be affected most by new hardware feature development since efficient changes or additions could not be made.

The principal reason the pre-4E2 fault recovery software exhibited this lack of flexibility was that it was functionally partitioned with a decentralized control structure. Figure 8 shows the functional partitions used in the pre-4E2 fault recovery software. They include:

(*i*) Peripheral Configuration Program,
(*ii*) Craft-Machine Interface Program,
(*iii*) Hardware Phase Recovery,
(*iv*) Error Analysis Program,
(*v*) Per unit fault recovery programs.

Each functional area contained control and unit-dependent code for many units embedded in the same programs. Each time a new unit was added, the functional area was modified, resulting in increased complexity, and requiring the entire area to be retested. Each one had decentralized control and had to provide for the following common requirements:

(*i*) Multilevel execution,
(*ii*) Unit dependent interfaces,

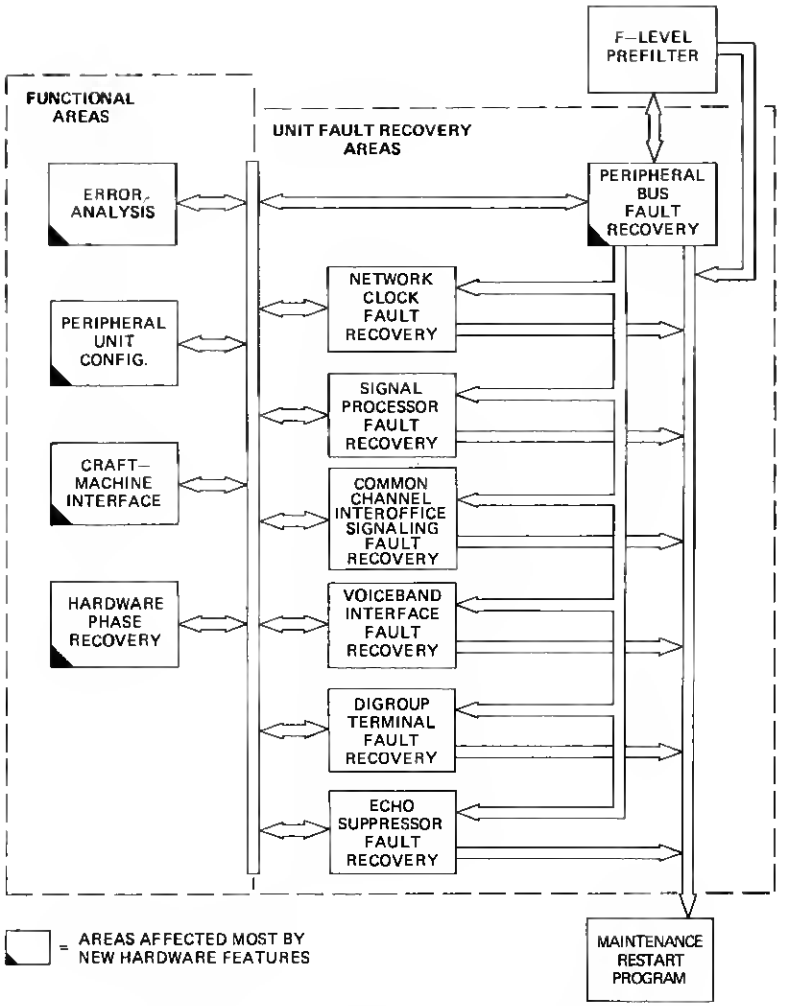

Fig. 8—Peripheral fault recovery structure (4E2 generic).

(*iii*) Hardware and software coordination,

(*iv*) Reentrant software.

Fault recovery software is required to execute on all system levels of execution, i.e., base level, BLM, interject, F-level interrupt, and phase level. In general, other maintenance software executes only on one level. In addition, fault recovery software is required to interface to all types of peripheral hardware. Hardware coordination is required because of the highly interconnected hardware. In particular, recovery from a problem in one unit generally affects other units. Software coordination is required to prevent software interaction due to time-

shared execution on base level. Reentrant software is required due to the multilevel execution of fault recovery software. As an example, the same fault recovery software may be started and successively restarted by escalating recovery actions. These requirements resulted in excessive interfaces and interaction between the different functional areas. Consequently, much fault recovery code was duplicated in an effort to reduce the number of interfaces. However, duplication made the job of maintaining the software much more difficult.

Each unit fault recovery program was responsible for many functions common to fault recovery of several units. The functions were being performed by several different programs in numerous ways. For example, each unit fault recovery program provided interfaces to each functional area, provided software to collect common recovery data, provided software to output recovery messages, etc.

Since most fault recovery software executes on interrupt level, it was designed with emphasis on real-time efficiency to minimize the interruption of base level due to a faulty unit. Techniques such as "tricky code" and private interfaces, as examples, were used for real-time efficiency. This also contributed to a structure that was difficult to change.

3.2.2 Development of improved structure

In response to these shortcomings in the pre-restructured fault recovery structure, an improved fault recovery structure was developed to incorporate the following: (*i*) a peripheral maintenance operating system; (*ii*) new hierarchically designed fault recovery control structures; (*iii*) a higher-level language; and (*iv*) a more formal development methodology.

The operating system would remove some complexity from the software by handling multilevel execution, memory allocation, and software coordination, and provide a truly standard interface between functional areas of fault recovery software.

The hierarchically designed control structure would provide complete separation between control and unit-dependent code. This would remove much of the unnecessary complexity in the control areas and limit the testing mainly to the new feature software being added. A hierarchical structure would lend itself more easily to changes and additions. It would improve readability and maintainability of the product.

A high-level language would improve programming productivity, readability, and maintainability. Programming productivity would be improved by allowing the programmer to concentrate on programming the function and not on initializing and saving registers, implementing loops, etc. Removing this level of detail from the source code would also improve the readability and maintainability of a program.

A formal development methodology would provide uniform and up-to-date documentation. The more rigorous steps in a methodology that insist on requirement reviews, design reviews, code walk-throughs, and test plan reviews help ensure that more software bugs are found early in the development. Other benefits of this formal development methodology, which uses development teams, are better project visibility and a larger group of people with knowledge of the software.

The development cost of the operating system and new control structures could be spread over several generics with little additional development cost beyond that required to add new units. Once the operating system and control structure were in place, the development costs for a new hardware-related feature would be reduced. In addition, program maintenance cost would be reduced.

Each of the above techniques, to some degree, has the drawback of less real-time efficiency and greater program size. The advantages stated above were judged to outweigh these considerations. It is also common practice when using a structured design approach to optimize after the design is working. Time should be scheduled for optimization when it can be determined which areas require real-time and program-size optimization. Note that optimization is generally easier in a structured design that is written in a high-level language. In many cases, large improvements in real-time and program store usage can be accomplished by small changes in a structure and/or compiler. Also, the increased program size is partially offset by reduced temporary memory requirements. This reduction can be attributed to more efficient use of temporary memory by the new operating system.

The fault recovery software, thus, evolved to a set of centralized control structures executing under a maintenance operating system. These control structures are designed with complete separation between control and unit-dependent code. Both maintenance control and unit-dependent software are written in EPLX. To add units to this new system, unit-dependent modules are added to each control structure as illustrated in Fig. 9 by the dashed blocks. In general, no modification is needed to the control structures. This results in testing the new unit software and little regression testing. In the pre-restructured fault recovery system each functional area required modifications to add the new control code and unit code. In Fig. 8 the blocks with the blacked-in corners are the functional areas that required extensive changes and additions. Each functional area required testing of the new unit fault recovery software and extensive regression testing of existing unit fault recovery software.

The new fault recovery system was planned to be evolved over several generics and to operate in parallel with the pre-restructure fault recovery system. The pre-restructure fault recovery system con-

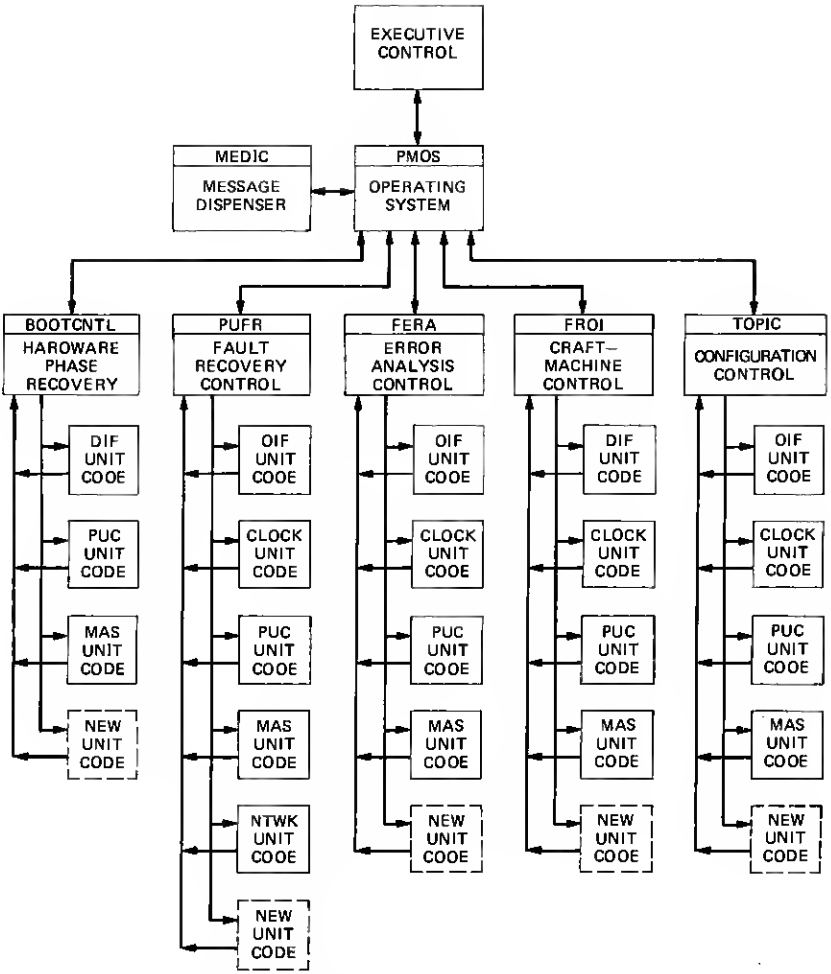

Fig. 9—Peripheral fault recovery structure (4E5 generic).

tinues to handle the units it was designed to accommodate. New units are being implemented under the new fault recovery system in the EPLX language.

3.3 Characterization of new fault recovery control architecture

The development of the new fault recovery control architecture was a multigeneric development. This new architecture was developed as a parallel system without disturbing the existing fault recovery software system, which supports the existing peripheral hardware architecture. The initial introduction was in the 4E3 generic with the development of the Peripheral Maintenance Operating System (PMOS)

and the Peripheral Unit Fault Recovery (PUFR) control structure. The first units supported by this system were the network frames (Time Slot Interchange and Time Multiplexed Switch). In the 4E5 generic, five new control structures were added, plus unit-dependent code for four new hardware frames. The five control structures were: (*i*) Toll Peripheral Configuration (TOPIC); (*ii*) Frame Request and Diagnostic Interface (FRDI); (*iii*) Failure Error Analysis (FERA); (*iv*) Message Dispenser and Coordinator (MEDIC); and (*v*) Bootstrap Control (BOOTCNTL) Program.

Each of the control structures, with the exception of MEDIC, was recommended as part of the original plan. Message Dispenser and Coordinator is a control structure which resulted from the introduction of intelligent (micropressor based) peripherals into the No. 4 ESS. These new peripherals execute macro-level orders which return multiword responses on a deferred basis after control is released. This required a new structure to control sending, receiving, and dispensing responses from these units. All four of the new hardware frames developed for 4E5 were of this type.

The resulting fault recovery software structure after the 4E5 generic is shown in Fig. 9. Each of the control structures was designed to meet the following objectives:

(*i*) Remove complex system dependencies by making use of the PMOS.

(*ii*) Make use of a more formal development methodology.

(*iii*) Use EPLX.

(*iv*) Provide complete separation between control and unit-dependent code.

(*v*) Provide standard unit-dependent interfaces.

(*vi*) Remove limitations (structure sizes, number of units, etc.) which exist in the present fault recovery software system.

(*vii*) Provide new capabilities.

Section 3.3.1 briefly describes each functional area of the new control architecture.

3.3.1 Peripheral maintenance operating system

The PMOS is the heart of the new fault recovery control architecture. This operating system centralizes peripheral maintenance control and coordination while reducing the complexity of system interaction. The operating system provides a standardized interface between PMOS tasks and the remainder of the system software. A PMOS task is a software process or function defined to the operating system. This interface allows an operating system task to be requested with several options specifying levels of execution, request mode, and priority. For example, simultaneous tasks can be scheduled on the same level or different levels, requested in a schedule and hold mode, parallel sched-

ule, or run-immediate mode. The operating system provides for task execution on base level, BLM, interject level, F-level, and phase level. Schedule and hold mode allows a task to schedule other tasks and be suspended until the other task is completed. Parallel schedule allows a task to schedule several other tasks and be suspended until all are completed. Run-immediate mode allows a task to request other tasks to be executed immediately. The main features of the operating system are: (*i*) task coordination; (*ii*) multilevel execution; (*iii*) administration of segment breaks; and (*iv*) reentry.

Peripheral fault recovery code for several units tends to be tightly coupled due to highly interconnected hardware units. The operating system removes from a task many of the concerns of task interaction by providing several task coordination functions. The operating system exercises blocking rules as defined in a central blocking table. Blocking prevents time-shared execution of specific tasks which otherwise would interact. Control of abort conditions and the execution of abort procedures are also provided. Control of execution and the determination of associated priorities are also included in a task coordination function.

Multilevel execution is a characteristic which in the past required numerous redundancies in many fault recovery programs. For example, each fault recovery program was required to check for the execution level and perform the necessary function to segment on that level. The operating system consolidates the necessary checks and functions to execute on different levels in one place. In general, tasks need not know what execution level they are on.

Segment breaks* required by base level processing add complexity and substantial development cost without a unified control architecture. Peripheral Maintenance Operating System provides segmentation routines for the new control structures. These routines preserve task memory when segment breaks are taken and ignore segment breaks on interrupt, interject, and phase level. The operating system also provides routines for timing breaks. Timing breaks at any execution level releases the operating system for execution of other tasks until the time specified at the break has expired. The task environment is preserved on segment breaks or timing breaks and reestablished upon return to the task.

Reentry is a condition that causes numerous problems for multilevel maintenance software. This problem arises, for example, when a multilevel program is interrupted on base level and the same program is entered on the interrupt. This can result in variables, initialized on base level, being overwritten on the interrupt level. This problem has

* Segment break is a convention in No. 4 ESS whereby all base level processing programs are required to return control to the Executive Control program every 3 ms.

forced fault recovery software to be exceedingly defensive during execution, adding system integrity checks and numerous other controls to avoid problems. The operating system resolves each case of reentry to the interrupted program. The integrity of each task is maintained either by aborting a task or allocating a different memory block to the task.

3.3.2 *Messaga dispenser and coordinator*

The MEDIC is a control structure developed to satisfy new requirements introduced with microprocessor-based frames in the peripheral system of the No. 4 ESS. Prior to 4E5, all peripheral frames on the peripheral unit bus returned responses to orders in the peripheral unit bus reply window. This window is 32 1A Processor cycles or 22.4 μs in duration. With the introduction of microprocessor-based frames, their macro-level orders required much longer times to complete because of the higher-level function being performed. These frames were designed to return an initial response in the reply window, indicating the order was accepted. A "task complete" response was returned when the macro work was completed within the frame.

Message Dispenser and Coordinator was developed as a control structure, having special interaction with the operating system. The basic functions of MEDIC are to (*i*) coordinate sending macro orders to microprocessor-based frames; (*ii*) poll these frames for responses on a deferred basis; and (*iii*) dispense those results to the appropriate client. The message dispenser, in conjunction with the operating system, provides primitives (low-level function calls) which allow a PMOS task to be suspended while waiting for a macro response. The task is automatically reactivated when the macro response is received. MEDIC provides a macro timeout notification. If a response is not received in a predefined maximum allowed time, the task is notified. The message dispenser also provides appropriate handling of unsolicited frame reports and autonomously generated reports. The unsolicited and autonomously generated reports are processed on BLM. The fault recovery program resolves the cause of the report and takes the appropriate recovery action to clear the problem.

3.3.3 *Paripheral unit fault recovery*

The PUFR control structure was the first control structure developed. It was developed in the 4E3 generic and supported the cost-reduced TSI frames. The Peripheral Unit Fault Recovery is a common control structure that handles all levels of peripheral error recovery (BLM, interject, and F-level). It consolidates common error-recovery functions under one control program by providing the following common functions: (*i*) initialization of data structures; (*ii*) collection of critical data required to isolate the source of a fault; (*iii*) an interface to unit-

dependent tasks to isolate the fault; (*iv*) an interface to error analysis programs to acknowledge the recovery actions; (*v*) execution of the necessary actions to recover the system; (*vi*) scheduling of any deferred maintenance actions, e.g., diagnostic, audits, etc.; and (*vii*) printing of reports containing critical data and the recovery action taken at the time of the fault.

The Peripheral Unit Fault Recovery controls the execution of fault recovery by calling both common routines and special unit dependent procedures. It satisfies the requirements of complete separation of control and unit-dependent software by providing standard interfaces to unit-dependent procedures. It calls the appropriate unit procedure by indexing a table based on unit identity. In addition to consolidating the common functions, PUFR also provides several enhancements. Some of these enhancements are: (*i*) multiple isolation attempts; (*ii*) multiple unit interface to error analysis; and (*iii*) enhanced report messages. Multiple isolation attempts allow a unit isolate program to request an isolation attempt on a different unit, for the same interrupt, when the problem cannot be resolved to tbe original unit. The subsequent isolation attempt is usually on a connecting frame. Multiple unit interface to error analysis allows the unit isolation program to pass a list of suspect units to error analysis when the problem cannot be resolved to a single unit. Enhanced report messages provide the craft with additional information concerning the source of the interrupt and the corrective action taken.

3.3.4 Failure error analysis

The FERA program provides the centralized control structure for carrying out the fault recovery error analysis function. The main role of error analysis in the No. 4 ESS is listed below:

(*i*) Complement fault recovery by adding the element of interrupt history,

(ii) Resolve intermittent and transient hardware faults,

(*iii*) Resolve faults in interconnected bardware,

(*iv*) Isolate persistent or intermittent system troubles in highly interconnected hardware subsystems,

(*v*) Record and analyze error history information,

(*vi*) Provide graceful degradation by removing units, which causes the minimum service effect, to correct a system problem,

(*vii*) Monitor deferred maintenance activities to guard against removing the redundant part of an intermittent failing piece of equipment.

Failure Error Analysis provides these functions by determining recovery actions with strategy tables. Strategy tables are a collection of decision schemes which make different decisions on successive occurrences of an error. A strategy table is selected by fault recovery

based on the type of fault occurring. The Analysis can acknowledge and accept the action recommended by PUFR or recommend an alternate action. The decision schemes and the selection of a strategy table use several factors to reach a decision:

(*i*) environment of the configurable portion of the system (simplex, duplex),

(*ii*) number of times the fault has occurred,

(*iii*) type of fault (unique, nonunique),

(*iv*) characterisic of the fault (transient, hard failure, illegal system action).

The Analysis also provides control for alternate recovery strategies. This control provides better analysis functions to be performed. Another new feature is a parallel analysis capability, which allows a strategy table and analysis function to execute in parallel. If the analysis function reaches a conclusion, it can override the action recommended by the strategy table. These new strategies allow FERA to resolve intermittent faults, transient faults in interconnected hardware, and persistent troubles more efficiently than tbe existing error analysis program. These enhancements were provided in addition to meeting the common objectives of all tbe new control structures. Secondary functions provided by FERA are (*i*) monitoring manual configuration requests; (*ii*) monitoring deferred maintenance actions (diagnostics, routine exercises); and (*iii*) manual input/output for control and display of FERA functions and data.

3.3.5 Craft-machine control program

The craft-machine interface functions are provided by the FRDI control structure. It provides the basic interface for manual configuration requests of the No. 4 ESS peripherals from either the TTY or Power Control Switch (PCS) located on the frame. Requests from the TTY may be for removal, restoral with diagnostic, restoral without diagnostics (unconditional), or for a switch of an active unit. Requests from the PCS may be for removal of a unit or restoral with a diagnostic. When any manual configuration requests are initiated, FRDI validates the request, interfaces with the TOPIC program to perform the configuration, interfaces with the FERA program to monitor the request, and prints the appropriate message to indicate whether the action was completed or denied. In addition to printing a message, if the request was initiated from a PCS, lights at the frame are lighted or extinguished to acknowledge the request.

The Frame Request and Diagnostic Interface is also the primary interface to the Diagnostic Control program for peripheral configuration before and after diagnostic requests. All diagnostic requests, independent of the source, are validated by FRDI. After the request is validated, the appropriate configuration function is requested. After

the diagnostic, FRDI controls the disposition of the unit by either restoring it or leaving it out of service. This decision depends upon a variety of conditions, such as the termination condition of the diagnostic, the results of the diagnostic, the type of request, and the state of the PCS.

3.3.6 *Configuration control*

The configuration control in the new peripheral fault recovery control architecture is provided by the TOPIC program. The Toll Peripheral Configuration program is responsible for establishing the configuration of the new peripherals introduced in the 4E5 generic. It is also responsible for the configuration of the Network Clock (NCLK) and System Clock (SYSCLK). These latter units existed in the initial release of the No. 4 ESS generic, 4E0. However, with the addition of the Network Clock Synchronization Unit (NCSU) in 4E5,[5] a major portion of the configuration software had to be modified and was moved into the new architecture.

Beyond the primary function of accepting requests from all sources and directing configuration requests to the specific unit-dependent program, TOPIC will determine if there are any connecting unit considerations, for example, clock or voice data path dependencies. If there are, TOPIC will take appropriate action on the connecting units. It also attempts to leave the resultant peripheral system configuration in a state that minimizes service degrading conditions. This is also based on connecting unit status.

3.3.7 *Bootstrap control*

Bootstrap is a function executed during phases of recovery which include hardware configuration. The bootstrap function for the new units in the 4E5 generic is controlled by the BOOTCNTL program. The function of bootstrap is to initialize the hardware and execute access tests. The degree of initialization and access testing varies depending on the phase of recovery (phases 1 to 4). With the introduction of microprocessor based units in 4E5, a large portion of the initialization of these frames is performed by firmware resident within the frame. During this initialization process, the 1A Processor is free to perform other functions. Bootstrap Control, making use of features provided by the operating system, is free to start the bootstrapping of other units. This technique is referred to as parallel bootstrap. This process results in less total time required to bootstrap several frames than would be required if the function were executed in a serial fashion.

Bootstrap Control also provides output containing the results of access tests performed during the phase. This information is useful to

the craft in understanding the final configuration after a phase, and in troubleshooting the peripherals removed from service during a phase.

3.4 A fault recovery example

This section presents a simple example of recovery from a hardware fault by the restructured fault recovery system. This example is provided to give a clear understanding of the function of each control structure. Figure 10 illustrates the actions taken in this example. A single hard (nonintermittent) fault in a duplicated unit is assumed for this example. The FRDI and BOOTCNTL structures are not involved in this example since they primarily execute on base and phase level, respectively.

A hardware error triggers an F-level interrupt and results in a PUFR F-level task being scheduled in PMOS. Peripheral Unit Fault Recovery performs initialization of internal data structures and collects data at the time of the interrupt which reflects the state of the system and the interrupting unit. It then schedules (schedule and hold mode) the unit's fault recovery task, passing the data it has gathered as input. Peripheral Unit Fault Recovery is suspended until the unit fault recovery task completes.

The unit fault recovery task will attempt to isolate the source of the interrupt to a configurable piece of hardware (half of a duplicated unit, etc.). The unit fault recovery task will analyze the input data, perform access tests, and reconfigure the unit and retry peripheral orders to isolate the source of the error. The suspect unit half, fault class (hard fault, software fault, intermittent fault, etc.), resolution class (resolved, unresolved), and recommended actions are returned to PUFR in a data block passed as input. Peripheral Unit Fault Recovery is reactivated when the unit fault recovery task is completed.

It then schedules (schedule and hold mode) the FERA task passing the suspect unit, fault class, and resolution class as inputs. The Analysis then determines if this is the first interrupt for this unit by consulting history data files. A new history data file is allocated, if it is the first interrupt. The Analysis updates a history data file, if a previous interrupt has been recorded for the suspect unit. It then selects a primary strategy based on interrupting unit, number of interrupts, fault class, resolution class, and configuration of the unit at the time of the error (simplex or duplex). The primary strategy acknowledges the action recommended by the unit fault recovery task or specifies an alternate action. The primary strategy is not used if the number of interrupts which has occurred on this unit is greater than the designed limits of a strategy table. The Analysis control selects a secondary strategy if this occurs. Otherwise, the secondary strategy is not used if a recovery action is specified by the primary strategy. The Analysis

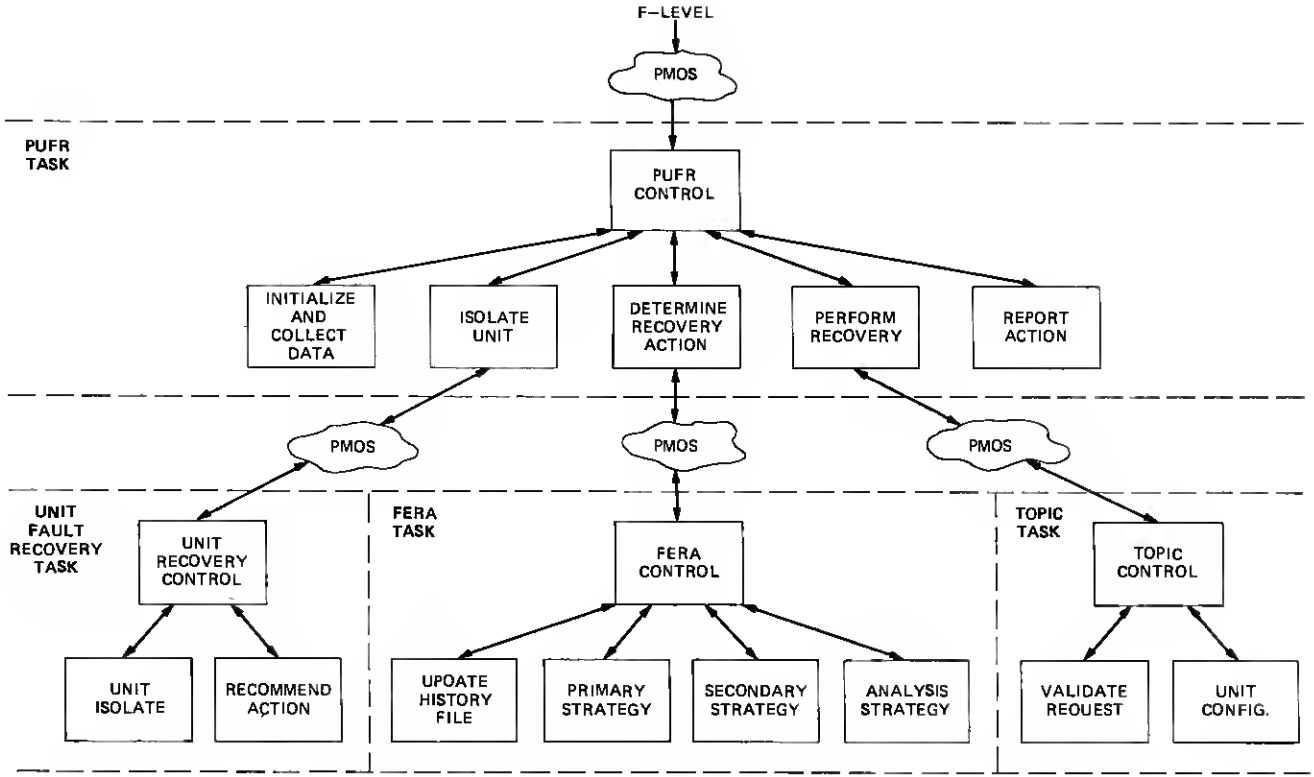

Fig. 10—Example fault recovery actions.

control always executes the analysis strategy independent of the action specified in the primary or secondary strategy. The analysis strategy views the interrupt in terms of a potential multi-unit hardware interconnection fault. The analysis strategy can overide the action specified by the primary or secondary strategy if an interconnection fault is suspected. The Analysis returns the recommended recovery action to PUFR in a data block passed as input.

The Peripheral Unit Fault Recovery is again reactivated when FERA completes. The recovery actions specified by FERA are then performed. The recovery actions may be an immediate action (configuration, etc.) or a deferred action (diagnostic, audits, etc.). Deferred actions are scheduled for base level execution. Immediate actions are scheduled for F-level execution. In the case of an immediate configuration action, PUFR schedules a TOPIC task in a schedule and hold mode. Toll Peripheral Configuration validates the configuration request and interfaces to the specified unit software for the function requested (remove, restore, switch, etc.). PUFR is reactivated after tbe TOPIC task is completed.

At this point the recovery is completed. The remaining function of the Peripheral Unit Fault Recovery is to format and output the data collected at the time of the interrupt and the recovery actions taken. It then returns to PMOS, completing the F-level processing.

The Peripheral Maintenance Operations System returns to base level processing after it determines that no other F-level tasks are scheduled. The deferred actions scheduled for base level are then executed.

3.5 Evaluation

With the release of the 4E5 generic, the multigeneric plan to develop a centralized peripheral fault-recovery-control architecture and a maintenance operating system is complete. This new control structure provides the fault recovery capability for four new peripheral unit types (DIF, PUC, MAS, NCSU).[5-7] Four other peripheral unit types (TSI, TMS, NCLK, SYSCLK) are partially supported under this system.

The development cost of the new fault recovery control architecture and the unit dependent code for the units listed above has been slightly larger than the original estimates. This difference can be partially attributed to the introduction of microprocessor technology with these units. It can also be said that introducing this new technology in the pre-restructured maintenance system would have resulted in even larger software development costs for these units.

The new fault-recovery-control architecture provides a well-documented, flexible-control architecture with well-defined interfaces to unit-dependent code. With this control architecture in place, new

features can be developed with an estimated 30 to 50 percent savings in fault recovery software effort. A portion of this savings can be attributed to the development methodology and the use of a high-level language. It is difficult to estimate the savings contributed by either factor. Also, it is not clear that the total benefit of either factor can be attained without the other also being present.

There are also many side benefits in addition to a decrease in development costs. Most of these benefits stem from the use of a modern development methodology. Improved, up-to-date documentation is one benefit already mentioned. Others are (*i*) better project visibility through the use of development teams and walkthroughs; (*ii*) a larger base of people with knowledge of specific software modules through the use of development teams; and (*iii*) more software bugs found early in the development prior to laboratory testing and field release.

Disadvantages of the new fault recovery control architecture and structure design methodology are increased program size and real-time usage. Real-time usage in error recovery, even though critical, does not significantly affect the system call handling capability as in call processing programs. These disadvantages were anticipated, but it was unclear what increase could be expected. Initial data indicate that a specific function like error recovery, which is real-time critical, has the following distribution of real-time usage:

Operating system—10 percent,
Control structures—2 percent,
Macro waits—35 percent,
Unit code—53 percent.

Some portion of the operating system, control structures, and unit code time can be attributed to the use of a high-level language. However, without recoding specific procedures it is difficult to determine what percentage is due to the language. Experiments have been performed comparing EPLX with the previously used language (EPL). In general, EPLX used more real time and program store than EPL. However, with some optimization the EPLX program was nearly as efficient as the EPL program. This indicates that there is little inherent inefficiency in the language. With proper knowledge of the language, programs can be optimized for both real time and program store usage.

IV. SUMMARY

This paper has described two specific examples which show how the No. 4 ESS has evolved through software restructure to better accommodate the addition of new hardware and software features. Call Processing and Fault Recovery software underwent varying degrees of incremental restructure. These software areas were considered for

restructure because many new features were to be added to the No. 4 ESS which directly affected Call Processing and Fault Recovery. The restructuring efforts focused on improving the deficiencies of the pre-restructure software and made use of modern development methodologies and a high-level programming language to accomplish the objectives. The resultant architectures are heirarchical, much more modular, and more easily modified and maintained. We acknowledge the effort of those designers too numerous to mention, who contributed to the successful Call Processing and Fault Recovery restructuring effort.

REFERENCES

1. E. A. Davis and P. K. Giloth, "No. 4 ESS Performance Objectives and Service Experience," B.S.T.J., this issue.
2. T. J. Cieslak et al., "Software Organization and Basic Call Handling," B.S.T.J., *56*, No. 7 (September 1977), pp. 1113–1138.
3. M. N. Meyers, W. A. Routt, and K. W. Yoder, "No. 4 ESS Maintenance Software," B.S.T.J., *56*, No. 7 (September 1977), pp. 1139–67.
4. "1A Processor," B.S.T.J., *56*, No. 2 (February 1977), pp. 119–327.
5. R. Metz and D. F. Winchell, "No. 4 ESS Network Clock Synchronization," B.S.T.J., this issue.
6. T. W. Anderson et al., "No. 4 ESS Mass Announcement Subsystem," B.S.T.J., this issue.
7. K. M. Hoppner et al., "No. 4 ESS Digital Interface," B.S.T.J., this issue.